17 Monitoring and troubleshooting errors with logging

This chapter covers

- Understanding the components of a log message
- Writing logs to multiple output locations
- Controlling log verbosity in different environments using filtering
- Using structured logging to make logs searchable

Logging is one of those topics that seems unnecessary, right up until you desperately need it! There's nothing more frustrating than finding an issue that you can only reproduce in production, and then discovering there are no logs to help you debug it.

Logging is the process of recording events or activities in an app, and it often involves writing a record to a console, a file, the Windows Event Log, or some other system. You can record anything in a log message, though there are generally two different types of messages:

- *Informational messages*—A standard event occurred: a user logged in, a product was placed in a shopping cart, or a new post was created on a blogging app.
- *Warnings and errors*—An error or unexpected condition occurred: a user had a negative total in the shopping cart, or an exception occurred.

Historically, a common problem with logging in larger applications was that each library and framework would generate logs in a slightly different format, if at all. When an error occurred in your app and you were trying to diagnose it, this inconsistency made it harder to connect the dots in your app to get the full picture and understand the problem.

Luckily, ASP.NET Core includes a new, generic logging interface that you can plug into. It's used throughout the ASP.NET Core framework code itself, as well as by third-party libraries, and you can easily use it to create logs in your own code. With the ASP.NET Core logging framework, you can control the verbosity of logs coming from each part of your code, including the framework and libraries, and you can write the log output to any destination that plugs into the framework.

In this chapter, I cover the ASP.NET Core logging framework in detail and explain how you can use it to record events and diagnose errors in your own apps. In section 17.1 I'll describe the architecture of the logging framework. You'll learn how DI makes it easy for both libraries and apps to create log messages, as well as to write those logs to multiple destinations.

In section 17.2 you'll learn how to write your own log messages in your apps with the `ILogger` interface. We'll break down the anatomy of a typical log record and look at its properties, such as the log level, category, and message.

Writing logs is only useful if you can read them, so in section 17.3 you'll learn how to add *logging providers* to your application. Logging providers control where your app writes your log messages. This could be to the console, to a file, or even to an external service. I'll show you how to add a logging provider that writes logs to a file, and how to configure a popular third-party logging provider called Serilog in your app.

Logging is an important part of any application, but determining *how much* logging is enough can be a tricky question. On the one hand, you want to provide sufficient information to be able to diagnose any problems. On the other, you don't want to fill your logs with data that makes it hard to find the important information when you need it. Even worse, what is sufficient in development might be far too much once you're running in production.

In section 17.4 I'll explain how you can filter log messages from various sections of your app, such as the ASP.NET Core infrastructure libraries, so that your logging providers only write the important messages. This lets you keep that balance between extensive logging in development and only writing important logs in production.

In the final section of this chapter, I'll touch on some of the benefits of *structured logging*, an approach to logging that you can use with some providers for the ASP.NET Core logging framework. Structured logging involves attaching data to log messages as key-value pairs to make it easier to search and query logs. You might attach a unique customer ID to every log message generated by your app, for example. Finding all the log messages associated with a user is much simpler with this approach, compared to recording the customer ID in an inconsistent manner as part of the log message.

We'll start this chapter by digging into what logging involves, and why your future self will thank you for using logging effectively in your application. Then we'll look at the pieces of the ASP.NET Core logging framework you'll use directly in your apps, and how they fit together.

17.1 Using logging effectively in a production app

Imagine you've just deployed a new app to production, when a customer calls saying that they're getting an error message using your app. How would you identify what caused the problem? You could ask the customer what steps they were taking, and potentially try to recreate the error yourself, but if that doesn't work, you're left trawling through the code, trying to spot errors with nothing else to go on.

Logging can provide the extra context you need to quickly diagnose a problem. Arguably, the most important logs capture the details about the error itself, but the events that led to the error can be just as useful in diagnosing the cause of an error.

There are many reasons for adding logging to an application, but, typically, the reasons fall into one of three categories:

- Logging for auditing or analytics reasons, to trace when events have occurred
- Logging errors
- Logging non-error events to provide a breadcrumb trail of events when an error does occur

The first of these reasons is simple. You may be required to keep a record of every time a user logs in, for example, or you may want to keep track of how many times a particular API method is called. Logging is an easy way to record the behavior of your app, by writing a message to the log every time an interesting event occurs.

I find the second reason for logging to be the most common. When an app is working perfectly, logs often go completely untouched. It's when there's an issue and a customer comes calling that logs become invaluable. A good set of logs can help you understand the conditions in your app that caused an error, including the context of the error itself, but also the context in previous requests.

> **TIP** Even with extensive logging in place, you may not realize you have an issue in your app unless you look through your logs regularly. For any medium to large app, this becomes impractical, so monitoring services such as Raygun (https://raygun.io) or Sentry (https://sentry.io) can be invaluable for notifying you of issues quickly.

If this sounds like a lot of work, then you're in luck. ASP.NET Core does a ton of the "breadcrumb" logging for you so that you can focus on creating high-quality log messages that provide the most value when diagnosing problems.

17.1.1 Highlighting problems using custom log messages

ASP.NET Core uses logging throughout its libraries. Depending on how you configure your app, you'll have access to the details of each request and EF Core query, even without adding additional logging messages to your own code. In figure 17.1 you can see the log messages created when you view a single recipe in the recipe application.

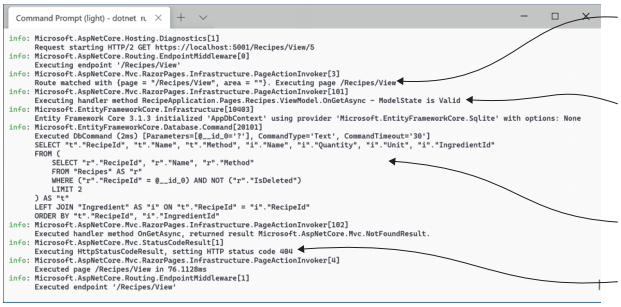


Figure 17.1 The ASP.NET Core Framework libraries use logging throughout. A single request generates multiple log messages that describe the flow of the request through your application.

This gives you a lot of useful information. You can see which URL was requested, the Razor Page and page handler that were invoked, the EF Core database command, the action result invoked, and the response. This information can be invaluable when trying to isolate a problem, whether a bug in a production app or a feature in development when you're working locally.

This infrastructure logging can be useful, but log messages that you create yourself can have even greater value. For example, you may be able to spot the cause of the error from the log messages in figure 17.1—we're attempting to view a recipe with an unknown `RecipeId` of `5`, but it's far from obvious. If you explicitly add a log message to your app when this happens, as in figure 17.2, the problem is much more apparent.

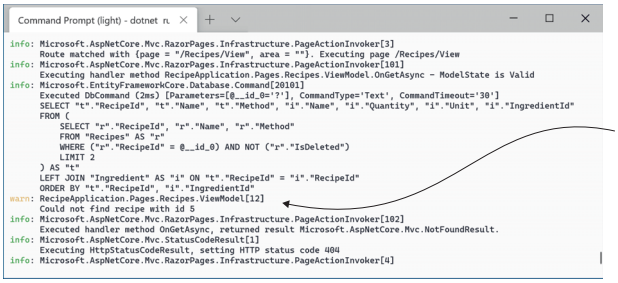


Figure 17.2 You can write your own logs. These are often more useful for identifying issues and interesting events in your apps.

This custom log message easily stands out and clearly states both the problem (the recipe with the requested ID doesn't exist) and the parameters/variables that led to the issue (the ID value of `5`). Adding similar log messages to your own applications will make it easier for you to diagnose problems, track important events, and generally know what your app is doing.

Hopefully you're now motivated to add logging to your apps, so we'll dig into the details of what that involves. In section 17.1.2 you'll see how to create a log message and how to define where the log messages are written. We'll look in detail at these two aspects in sections 17.2 and 17.3; first, though, we'll look at where they fit in terms of the ASP.NET Core logging framework as a whole.

17.1.2 The ASP.NET Core logging abstractions

The ASP.NET Core logging framework consists of a number of logging abstractions (interfaces, implementations, and helper classes), the most important of which are shown in figure 17.3:

- `ILogger`—This is the interface you'll interact with in your code. It has a `Log()` method, which is used to write a log message.
- `ILoggerProvider`—This is used to create a custom instance of an `ILogger`, depending on the provider. A "console" `ILoggerProvider` would create an `ILogger` that writes to the console, whereas a "file" `ILoggerProvider` would create an `ILogger` that writes to a file.
- `ILoggerFactory`—This is the glue between the `ILoggerProvider` instances and the `ILogger` you use in your code. You register `ILoggerProvider` instances with an `ILoggerFactory` and call `CreateLogger()` on the `ILoggerFactory` when you need an `ILogger`. The factory creates an `ILogger` that wraps each of the providers, so when you call the `Log()` method, the log is written to every provider.

The design in figure 17.3 makes it easy to add or change where your application writes the log messages, without having to change your application code. The following listing shows all the code required to add an `ILoggerProvider` that writes logs to the console.

Listing 17.1 Adding a console log provider to `IHost` in Program.cs

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        new HostBuilder()
            .ConfigureLogging(builder =>builder.AddConsole())
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Add new providers with the ConfigureLogging extension method on HostBuilder.

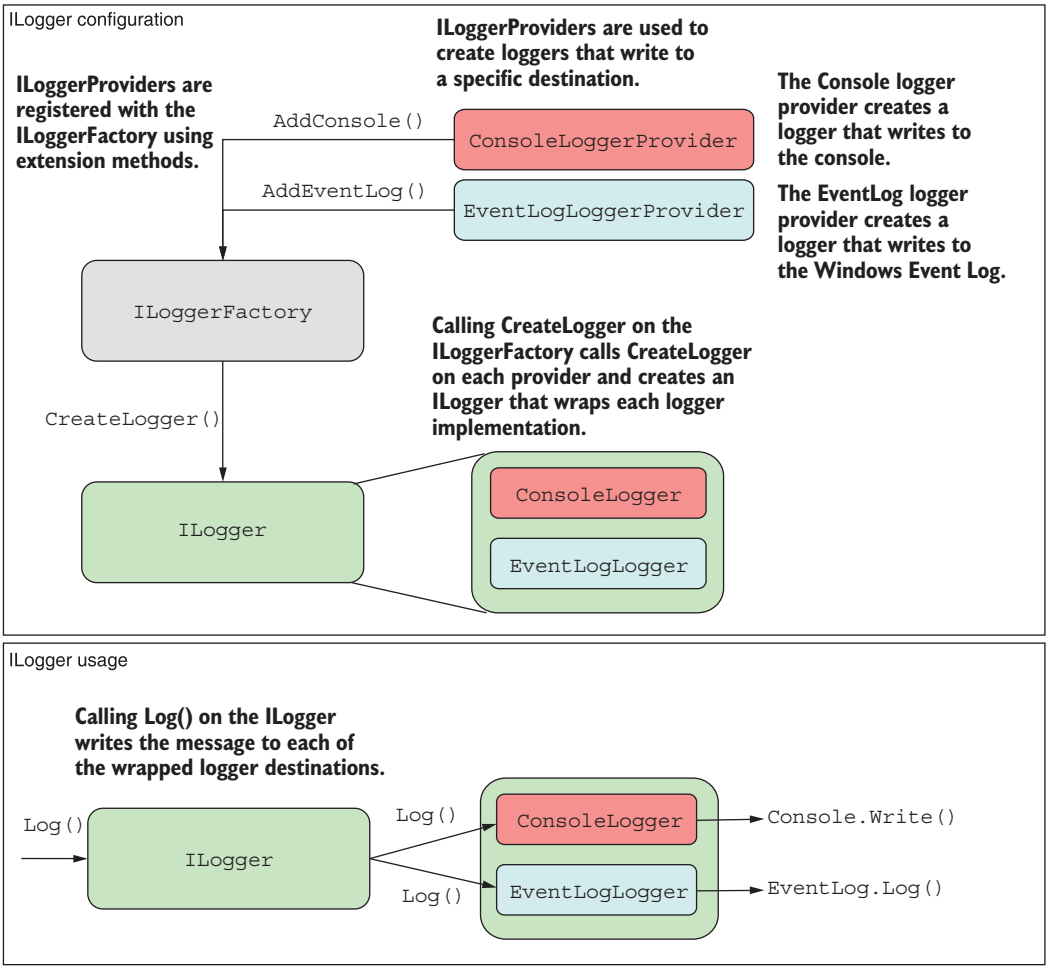


Figure 17.3 The components of the ASP.NET Core logging framework. You register logging providers with an `ILoggerFactory`, which is used to create implementations of `ILogger`. You write logs to the `ILogger`, which uses the `ILogger` implementations to output logs to the console or a file. This design allows you to send logs to multiple locations without having to configure those locations when you create a log message.

> **NOTE** The console logger is added by default in the `CreateDefaultBuilder` method, as you'll see in section 17.3.

Other than this configuration on `IHostBuilder`, you don't interact with `ILoggerProvider` instances directly. Instead, you write logs using an instance of `ILogger`, as you'll see in the next section.

17.2 Adding log messages to your application

In this section we'll look in detail at how to create log messages in your own application. You'll learn how to create an instance of `ILogger`, and how to use it to add logging to an existing application. Finally, we'll look at the properties that make up a logging record, what they mean, and what you can use them for.

Logging, like almost everything in ASP.NET Core, is available through DI. To add logging to your own services, you only need to inject an instance of `ILogger<T>`, where `T` is the type of your service.

> **NOTE** When you inject `ILogger<T>`, the DI container indirectly calls `ILoggerFactory.CreateLogger<T>()` to create the wrapped `ILogger` of figure 17.3. In section 17.2.2 you'll see how to work directly with `ILoggerFactory` if you prefer. The `ILogger<T>` interface also implements the non-generic `ILogger` interface but adds additional convenience methods.

You can use the injected `ILogger` instance to create log messages, which it writes to each configured `ILoggerProvider`. The following listing shows how to inject an `ILogger<>` instance into the `PageModel` of the Index.cshtml Razor Page for the recipe application from previous chapters and how to write a log message indicating how many recipes were found.

Listing 17.2 Injecting `ILogger` into a class and writing a log message

```
public class IndexModel : PageModel
{
    private readonly RecipeService _service;
    private readonly ILogger<IndexModel> _log;

    public ICollection<RecipeSummaryViewModel> Recipes { get; set; }

    public IndexModel(
        RecipeService service,
        ILogger<IndexModel> log)
    {
        _service = service;
        _log = log;
    }

    public void OnGet()
    {
        Recipes = _service.GetRecipes();
        _log.LogInformation(
            "Loaded {RecipeCount} recipes", Recipes.Count);
    }
}
```

Injects the generic ILogger<T> using DI, which implements ILogger

This writes an Information-level log. The RecipeCount variable is substituted in the message.

In this example, you're using one of the many extension methods on `ILogger` to create the log message, `LogInformation()`. There are many extension methods on `ILogger` that let you easily specify a `LogLevel` for the message.

DEFINITION The *log level* of a log is how important it is and is defined by the `LogLevel` enum. Every log message has a log level.

You can also see that the message you pass to the `LogInformation` method has a placeholder indicated by braces, `{RecipeCount}`, and you pass an additional parameter, `Recipes.Count`, to the logger. The logger will replace the placeholder with the parameter at runtime. Placeholders are matched with parameters by position, so if you include two placeholders, for example, the second placeholder is matched with the second parameter.

TIP You could have used normal string interpolation to create the log message; for example, `$"Loaded {Recipes.Count} recipes"`. But I recommend always using placeholders, as they provide additional information for the logger that can be used for structured logging, as you'll see in section 17.5.

When the `OnGet` page handler in the `IndexModel` executes, `ILogger` writes a message to any configured logging providers. The exact format of the log message will vary from provider to provider, but figure 17.4 shows how the console provider would display the log message from listing 17.2.

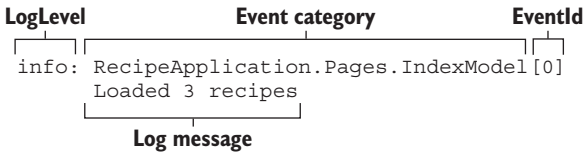


Figure 17.4 An example log message, as it's written to the default console provider. The log level category provides information about how important the message is and where it was generated. The EventId provides a way to identify similar log messages.

The exact presentation of the message will vary depending on where the log is written, but each log record includes up to six common elements:

- *Log level*—The log level of the log is how important it is and is defined by the `LogLevel` enum.
- *Event category*—The category may be any string value, but it's typically set to the name of the class creating the log. For `ILogger<T>`, the full name of the type `T` is the category.
- *Message*—This is the content of the log message. It can be a static string, or it can contain placeholders for variables, as shown in listing 17.2. Placeholders are indicated by braces, {}, and are substituted with the provided parameter values.
- *Parameters*—If the message contains placeholders, they're associated with the provided parameters. For the example in listing 17.2, the value of `Recipes.Count` is

assigned to the placeholder called `RecipeCount`. Some loggers can extract these values and expose them in your logs, as you'll see in section 17.5.

- *Exception*—If an exception occurs, you can pass the exception object to the logging function along with the message and other parameters. The logger will log the exception, in addition to the message itself.
- *EventId*—This is an optional integer identifier for the error, which can be used to quickly find all similar logs in a series of log messages. You might use an `EventId` of `1000` when a user attempts to load a non-existent recipe, and an `EventId` of `1001` when a user attempts to access a recipe they don't have permission to access. If you don't provide an `EventId`, the value `0` will be used.

Not every log message will have all these elements—you won't always have an `Exception` or parameters, for example. There are various overloads to the logging methods that take these elements as additional method parameters. Besides these optional elements, each message will have, at the very least, a level, category, and message. These are the key features of the log, so we'll look at each in turn.

17.2.1 Log level: How important is the log message?

Whenever you create a log using `ILogger`, you must specify the *log level*. This indicates how serious or important the log message is, and it's an important factor when it comes to filtering which logs get written by a provider, as well as finding the important log messages after the fact.

You might create an `Information` level log when a user starts to edit a recipe. This is useful for tracing the application's flow and behavior, but it's not important, because everything is normal. But if an exception is thrown when the user attempts to save the recipe, you might create a `Warning` or `Error` level log.

The log level is typically set by using one of several extension methods on the `ILogger` interface, as shown in listing 17.3. This example creates an `Information` level log when the `View` method executes, and a `Warning` level error if the requested recipe isn't found.

Listing 17.3 Specifying the log level using extension methods on `ILogger`

```
private readonly ILogger _log;
public async IActionResult OnGet(int id)
{
    _log.LogInformation(
        "Loading recipe with id {RecipeId}", id);

    Recipe = _service.GetRecipeDetail(id);
    if (Recipe is null)
    {
        _log.LogWarning(
            "Could not find recipe with id {RecipeId}", id);
        return NotFound();
    }
```

An ILogger instance is injected into the controller using constructor injection.

Writes an Information level log message

Writes a Warning level log message

```
    return Page();
}
```

The `LogInformation` and `LogWarning` extension methods create log messages with a log level of `Information` and `Warning`, respectively. There are six log levels to choose from, ordered here from most to least serious:

- `Critical`—For disastrous failures that may leave the app unable to function correctly, such as out-of-memory exceptions or if the hard drive is out of disk space or the server is on fire.
- `Error`—For errors and exceptions that you can't handle gracefully; for example, exceptions thrown when saving an edited entity in EF Core. The operation failed, but the app can continue to function for other requests and users.
- `Warning`—For when an unexpected or error condition arises that you can work around. You might log a `Warning` for handled exceptions, or when an entity isn't found, as in listing 17.3.
- `Information`—For tracking normal application flow; for example, logging when a user logs in, or when they view a specific page in your app. Typically these log messages provide context when you need to understand the steps leading up to an error message.
- `Debug`—For tracking detailed information that's particularly useful during development. Generally this only has short-term usefulness.
- `Trace`—For tracking very detailed information, which may contain sensitive information like passwords or keys. It's rarely used, and not used at all by the framework libraries.

Think of these log levels in terms of a pyramid, as shown in figure 17.5. As you progress down the log levels, the importance of the messages goes down, but the frequency goes up. Generally you'll find many `Debug` level log messages in your application, but (hopefully) few `Critical` or `Error` level messages.

This pyramid shape will become more meaningful when we look at filtering in section 17.4. When an app is in production, you typically don't want to record all the `Debug` level messages generated by your application. The sheer volume of messages would be overwhelming to sort through and could end up filling your disk with messages that say, "Everything's OK!" Additionally, `Trace` messages shouldn't be enabled in production, as they may leak sensitive data. By filtering out the lower log levels, you can ensure that you generate a sane number of logs in production but have access to all the log levels in development.

In general, logs of a higher level are more important than lower-level logs, so a `Warning` log is more important than an `Information` log, but there's another aspect to consider. Where the log came from, or who created the log, is a key piece of information that's recorded with each log message and is called the *category*.

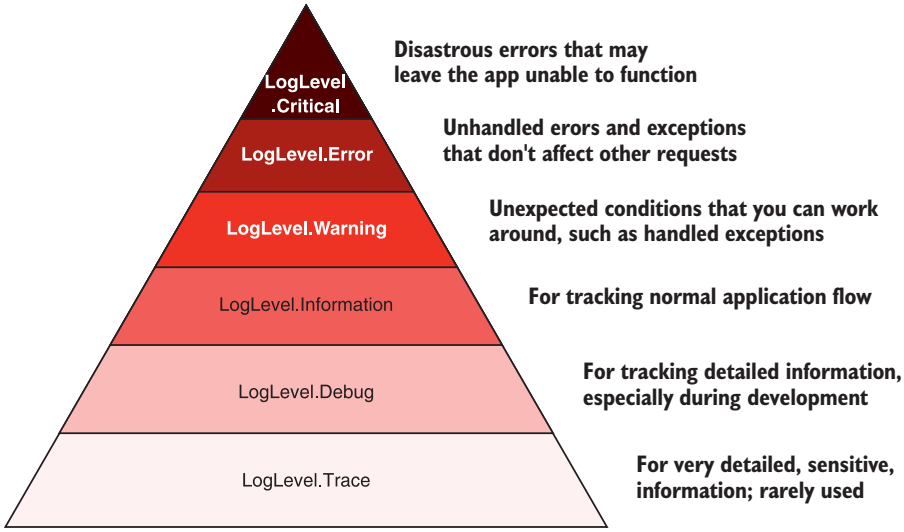


Figure 17.5 The pyramid of log levels. Logs with a level near the base of the pyramid are used more frequently but are less important. Logs with a level near the top should be rare but are important.

17.2.2 Log category: Which component created the log

As well as a log level, every log message also has a *category*. You set the log level independently for every log message, but the category is set when you create the `ILogger` instance. Like log levels, the category is particularly useful for filtering, as you'll see in section 17.4. It's written to every log message, as shown in figure 17.6.

Every log message has an associated category.

The category is typically set to the name of the class creating the log message.

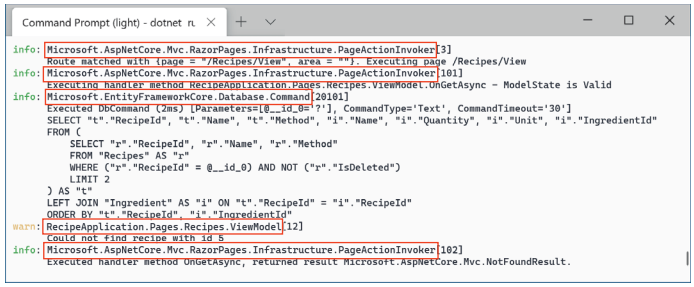


Figure 17.6 Every log message has an associated category, which is typically the class name of the component creating the log. The default console logging provider outputs the log category for every log.

The category is a `string`, so you can set it to anything, but the convention is to set it to the fully qualified name of the type that's using `ILogger`. In section 17.2 I achieved

this by injecting `ILogger<T>` into `RecipeController`; the generic parameter `T` is used to set the category of the `ILogger`.

Alternatively, you can inject `ILoggerFactory` into your methods and pass an explicit category when creating an `ILogger` instance. This lets you change the category to an arbitrary string.

Listing 17.4 Injecting `ILoggerFactory` to use a custom category

```
public class RecipeService
{
    private readonly ILogger _log;
    public RecipeService(ILoggerFactory factory)
    {
        _log = factory.CreateLogger("RecipeApp.RecipeService");
    }
}
```

Injects an ILoggerFactory instead of an ILogger directly

Passes a category as a string when calling CreateLogger

There is also an overload of `CreateLogger()` with a generic parameter that uses the provided class to set the category. If the `RecipeService` in listing 17.4 was in the `RecipeApp` namespace, the `CreateLogger` call could be written equivalently as

```
_log = factory.CreateLogger<RecipeService>();
```

Similarly, the final `ILogger` instance created by this call would be the same as if you'd directly injected `ILogger<RecipeService>` instead of `ILoggerFactory`.

TIP Unless you're using heavily customized categories for some reason, favor injecting `ILogger<T>` into your methods over `ILoggerFactory`.

The final compulsory part of every log entry is fairly obvious: the *log message*. At the simplest level, this can be any string, but it's worth thinking carefully about what information would be useful to record—anything that will help you diagnose issues later on.

17.2.3 Formatting messages and capturing parameter values

Whenever you create a log entry, you must provide a *message*. This can be any string you like, but as you saw in listing 17.2, you can also include placeholders indicated by braces, {}, in the message string:

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

Including a placeholder and a parameter value in your log message effectively creates a key-value pair, which some logging providers can store as additional information associated with the log. The previous log message would assign the value of `Recipes.Count` to a key, `RecipeCount`, and the log message itself is generated by replacing the placeholder with the parameter value, to give the following (where `Recipes.Count=3`):

```
"Loaded 3 recipes"
```

You can include multiple placeholders in a log message, and they'll be associated with the additional parameters passed to the log method. The order of the placeholders in the format string must match the order of the parameters you provide.

> **WARNING** You must pass at least as many parameters to the log method as there are placeholders in the message. If you don't pass enough parameters, you'll get an exception at runtime.

For example, the log message

```
_log.LogInformation("User {UserId} loaded recipe {RecipeId}", 123, 456)
```

would create the parameters `UserId=123` and `RecipeId=456`. *Structured logging* providers could store these values, in addition to the formatted log message `"User 123 loaded recipe 456"`. This makes it easier to search the logs for a particular `UserId` or `RecipeId`.

> **DEFINITION** *Structured or semantic logging* attaches additional structure to log messages to make them more easily searchable and filterable. Rather than storing only text, it stores additional contextual information, typically as key-value pairs. JSON is a common format used for structured log messages, as it has all of these properties.

Not all logging providers use semantic logging. The default console logging provider doesn't, for example—the message is formatted to replace the placeholders, but there's no way of searching the console by key-value.

But even if you're not using structured logging initially, I recommend writing your log messages as though you are, with explicit placeholders and parameters. That way, if you decide to add a structured logging provider later, you'll immediately see the benefits. Additionally, I find that thinking about the parameters that you can log in this way prompts you to record more parameter values, instead of only a log message. There's nothing more frustrating than seeing a message like `"Cannot insert record due to duplicate key"` but not having the key value logged!

> **TIP** Generally speaking, I'm a fan of C# 6's interpolated strings, but don't use them for your log messages when a placeholder and parameter would also make sense. Using placeholders instead of interpolated strings will give you the same output message but will also create key-value pairs that can be searched later.

We've looked a lot at how you can create log messages in your app, but we haven't focused on where those logs are written. In the next section we'll look at the built-in ASP.NET Core logging providers, how they're configured, and how you can replace the defaults with a third-party provider.

17.3 Controlling where logs are written using logging providers

In this section you'll learn how to control where your log messages are written by adding additional `ILoggerProviders` to your application. As an example, you'll see how to add a simple file logger provider that writes your log messages to a file, in addition to the existing console logger provider. In section 17.3.2 you'll learn how to swap out the default logging infrastructure entirely for an alternative implementation using the open source Serilog library.

Up to this point, we've been writing all our log messages to the console. If you've run any ASP.NET Core sample apps locally, you'll have probably already seen the log messages written to the console window.

> **NOTE** If you're using Visual Studio and debugging using the IIS Express option (the default), you won't see the console window (though the log messages are written to the Debug Output window instead). For that reason, I normally ensure I select the app name from the drop-down list in the debug toolbar, instead of IIS Express.

Writing log messages to the console is great when you're debugging, but it's not much use for production. No one's going to be monitoring a console window on a server, and the logs wouldn't be saved anywhere or be searchable. Clearly, you'll need to write your production logs somewhere else.

As you saw in section 17.1, *logging providers* control the destination of your log messages in ASP.NET Core. They take the messages you create using the `ILogger` interface and write them to an output location, which varies depending on the provider.

> **NOTE** This name always gets to me—the log *provider* effectively *consumes* the log messages you create and outputs them to a destination. You can probably see the origin of the name from figure 17.3, but I still find it somewhat counterintuitive.

Microsoft has written several first-party log providers for ASP.NET Core that are available out-of-the-box in ASP.NET Core. These include

- *Console provider*—Writes messages to the console, as you've already seen.
- *Debug provider*—Writes messages to the debug window when you're debugging an app in Visual Studio or Visual Studio Code, for example.
- *EventLog provider*—Writes messages to the Windows Event Log. Only outputs log messages when running on Windows, as it requires Windows-specific APIs.
- *EventSource provider*—Writes messages using Event Tracing for Windows (ETW) or LTTng tracing on Linux.

There are also many third-party logging provider implementations, such as an Azure App Service provider, an elmah.io provider, and an Elasticsearch provider. On top of that, there are integrations with other pre-existing logging frameworks like NLog and

Serilog. It's always worth looking to see whether your favorite .NET logging library or service has a provider for ASP.NET Core, as most do.

You configure the logging providers for your app in Program.cs using `HostBuilder`. The `CreateDefaultBuilder` helper method configures the console and debug providers for your application automatically, but it's likely you'll want to change or add to these.

You have two options when you need to customize logging for your app:

- Use your own `HostBuilder` instance, instead of `Host.CreateDefaultBuilder`, and configure it explicitly.
- Add an additional `ConfigureLogging` call after `CreateDefaultBuilder`.

If you only need to customize logging, the latter approach is simpler. But if you find you also want to customize the other aspects of the `HostBuilder` created by `CreateDefaultBuilder` (such as your app configuration settings), it may be worth ditching the `CreateDefaultBuilder` method and creating your own instance instead.

In section 17.3.1 I'll show how to add a simple third-party logging provider to your application that writes log messages to a file, so that your logs are persisted. In section 17.3.2 I'll show how to go one step further and replace the default `ILoggerFactory` in ASP.NET Core with an alternative implementation using the popular open source Serilog library.

17.3.1 Adding a new logging provider to your application

In this section we're going to add a logging provider that writes to a rolling file, so our application writes logs to a new file each day. We'll continue to log using the console and debug providers as well, because they will be more useful than the file provider when developing locally.

To add a third-party logging provider in ASP.NET Core, follow these steps:

1. Add the logging provider NuGet package to the solution. I'm going to be using a provider called NetEscapades.Extensions.Logging.RollingFile, which is available on NuGet and GitHub. You can add it to your solution using the NuGet Package Manager in Visual Studio, or using the .NET CLI by running

   ```
   dotnet add package NetEscapades.Extensions.Logging.RollingFile
   ```

   from your application's project folder.

   NOTE This package is a simple file logging provider, available at https://github.com/andrewlock/NetEscapades.Extensions.Logging. It's based on the Azure App Service logging provider. If you would like more control over your logs, such as specifying the file format, consider using Serilog instead, as described in section 17.3.2.

2. Add the logging provider using the `IHostBuilder.ConfigureLogging()` extension method. You can add the file provider by calling `AddFile()`, as shown in

the next listing. `AddFile()` is an extension method provided by the logging provider package to simplify adding the provider to your app.

Listing 17.5 Adding a third-party logging provider to `IHostBuilder`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(builder => builder.AddFile())
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The CreateDefaultBuilder method configures the console and debug providers as normal.

Adds the new file logging provider to the logger factory

NOTE Adding a new provider doesn't replace existing providers. Listing 17.5 uses the `CreateDefaultBuilder` helper method, so the console and debug logging providers have already been added. To remove them, call `builder.ClearProviders()` at the start of the `ConfigureLogging` method, or use a custom `HostBuilder`.

With the file logging provider configured, you can run the application and generate logs. Every time your application writes a log using an `ILogger` instance, `ILogger` writes the message to all configured providers, as shown in figure 17.7. The console messages are conveniently available, but you also have a persistent record of the logs stored in a file.

TIP By default, the rolling file provider will write logs to a subdirectory of your application. You can specify additional options such as filenames and file size limits using overloads of `AddFile()`. For production, I recommend using a more established logging provider, such as Serilog.

The key takeaway from listing 17.5 is that the provider system makes it easy to integrate existing logging frameworks and providers with the ASP.NET Core logging abstractions. Whichever logging provider you choose to use in your application, the principles are the same: call `ConfigureLogging` on `IHostBuilder` and add a new logging provider using extension methods like `AddConsole()`, or `AddFile()` in this case.

Logging your application messages to a file can be useful in some scenarios, and it's certainly better than logging to a non-existent console window in production, but it may still not be the best option.

If you discovered a bug in production and you needed to quickly look at the logs to see what happened, for example, you'd need to log on to the remote server, find the

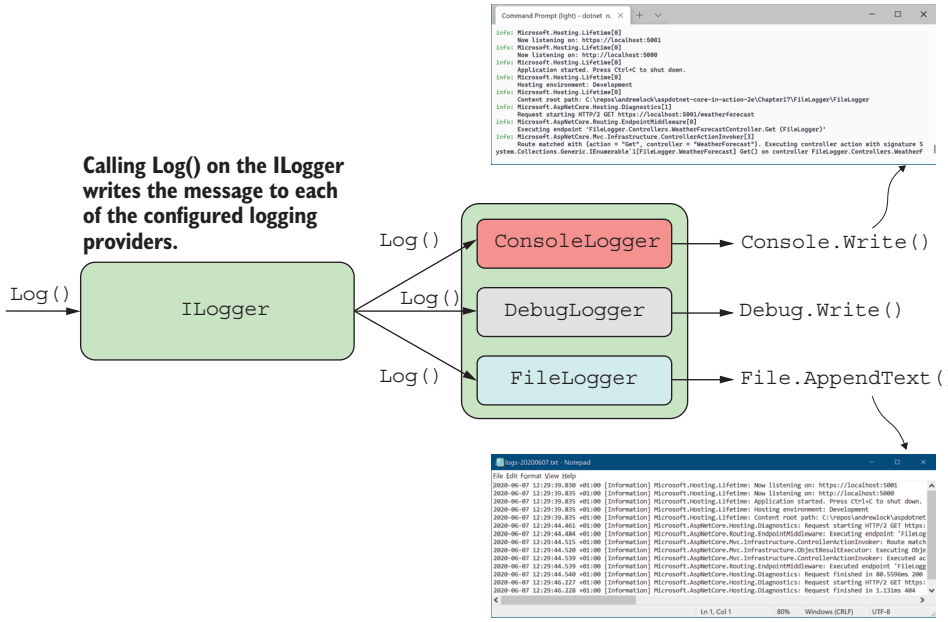


Figure 17.7 Logging a message with `ILogger` writes the log using all of the configured providers. This lets you, for example, log a convenient message to the console while also persisting the logs to a file.

log files on disk, and trawl through them to find the problem. If you have multiple web servers, you'd have a mammoth job to fetch all the logs before you could even start to tackle the bug. Not fun. Add to that the possibility of file permission or drive space issues, and file logging seems less attractive.

Instead, it's often better to send your logs to a centralized location, separate from your application. Exactly where this location may be is up to you; the key is that each instance of your app sends its logs to the same location, separate from the app itself.

If you're running your app on Azure, you get centralized logging for free because you can collect logs using the Azure App Service provider. Alternatively, you could send your logs to a third-party log aggregator service such as Loggr (http://loggr.net/), elmah.io (https://elmah.io/), or Seq (https://getseq.net/). You can find ASP.NET Core logging providers for each of these services on NuGet, so adding them is the same process as adding the file provider you've seen already.

Another popular option is to use the open source Serilog library to simultaneously write to a variety of different locations. In the next section I'll show how you can replace the default `ILoggerFactory` implementation with Serilog in your application, opening up a wide range of possible options for where your logs are written.

17.3.2 Replacing the default ILoggerFactory with Serilog

In this section we'll replace the default `ILoggerFactory` in the recipe app with an implementation that uses Serilog. Serilog (https://serilog.net) is an open source project that can write logs to many different locations, such as files, the console, an Elasticsearch cluster,[1] or a database. This is similar to the functionality you get with the default `ILoggerFactory`, but due to the maturity of Serilog, you may find you can write to more places.

Serilog predates ASP.NET Core, but thanks to the logging abstractions around `ILoggerFactory` and `ILoggerProvider`, you can easily integrate with Serilog while still using the `ILogger` abstractions in your application code.

Serilog uses a similar design philosophy to the ASP.NET Core logging abstractions—you write logs to a central logging object, and the log messages are written to multiple locations, such as the console or a file. Serilog calls each of these locations a *sink*.[2]

When you use Serilog with ASP.NET Core, you'll typically replace the default `ILoggerFactory` with a custom factory that contains a single logging provider, `SerilogLoggerProvider`. This provider can write to multiple locations, as shown in figure 17.8. This configuration is a bit of a departure from the standard ASP.NET Core logging setup, but it prevents Serilog's features from conflicting with equivalent features of the default `LoggerFactory`, such as filtering (see section 17.4).

> **TIP** If you're familiar with Serilog, you can use the examples in this section to easily integrate a working Serilog configuration with the ASP.NET Core logging infrastructure.

In this section we'll add a single sink to write the log messages to the console, but using the Serilog logging provider instead of the built-in console provider. Once you've set this up, adding additional sinks to write to other locations is trivial. Adding the Serilog logging provider to an application involves three steps:

1. Add the required Serilog NuGet packages to the solution.
2. Create a Serilog logger and configure it with the required sinks.
3. Call `UseSerilog()` on `IHostBuilder` to replace the default `ILoggerFactory` implementation with `SerilogLoggerFactory`. This configures the Serilog provider automatically and hooks up the already-configured Serilog logger.

To install Serilog into your ASP.NET Core app, you need to add the base NuGet package and the NuGet packages for any sinks you need. You can do this through the Visual Studio NuGet GUI, using the PMC, or using the .NET CLI. To add the

---

[1] Elasticsearch is a REST-based search engine that's often used for aggregating logs. You can find out more at www.elastic.co/elasticsearch/.

[2] For a full list of available sinks, see https://github.com/serilog/serilog/wiki/Provided-Sinks. There are 93 different sinks at the time of writing!

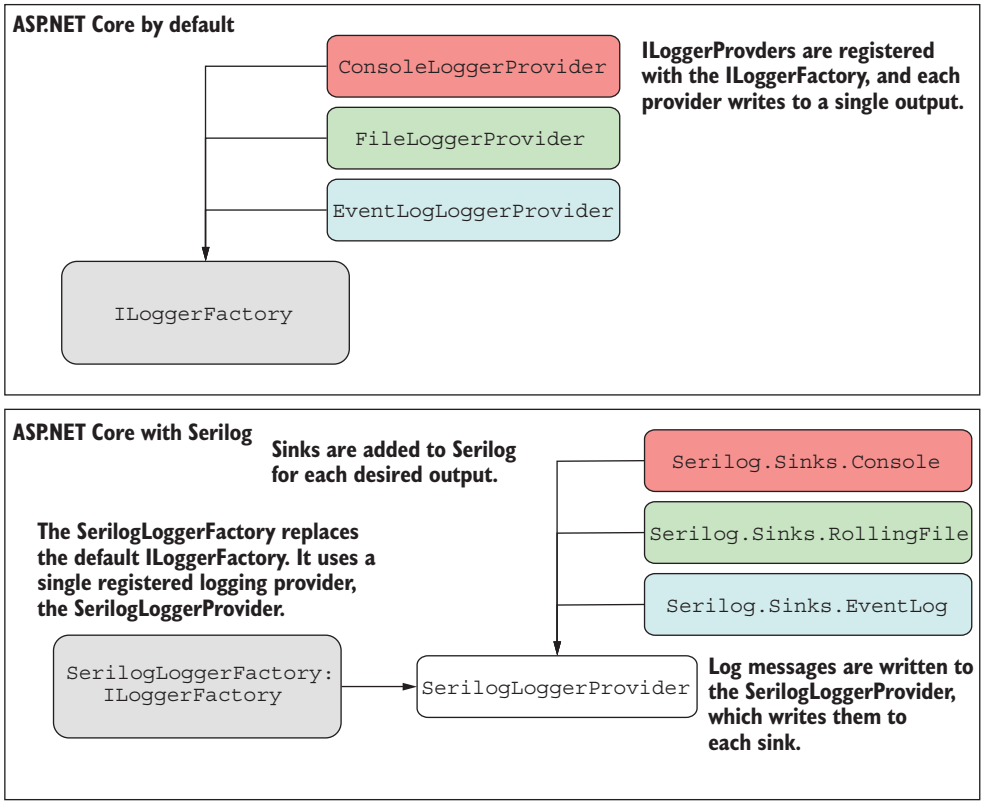


Figure 17.8 Configuration when using Serilog with ASP.NET Core compared to the default logging configuration. You can achieve the same functionality with both approaches, but you may find Serilog provides additional libraries for adding extra features.

Serilog ASP.NET Core package and a sink for writing to the console, run these commands:

```
dotnet add package Serilog.AspNetCore
dotnet add package Serilog.Sinks.Console
```

This adds the necessary NuGet packages to your project file and restores them. Next, create a Serilog logger and configure it to write to the console by adding the console sink, as shown in listing 17.6. This is the most basic of Serilog configurations, but you can add extra sinks and configuration here too.[3] I've also added a `try-catch-finally` block around our call to `CreateHostBuilder`, to ensure that logs are still written if

[3] You can customize Serilog until the cows come home, so it's worth consulting the documentation to see what's possible. The wiki is particularly useful: https://github.com/serilog/serilog/wiki/Configuration-Basics.

there's an error starting up the web host or there's a fatal exception. Finally, the Serilog logger factory is configured by calling `UseSerilog()` on the `IHostBuilder`.

Listing 17.6 Configuring a Serilog logging provider to use a console sink

```
public class Program
{
    public static void Main(string[] args)
    {
        Log.Logger = new LoggerConfiguration()
            .WriteTo.Console()
            .CreateLogger();
        try
        {
            CreateHostBuilder(args).Build().Run();
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Host terminated unexpectedly");
        }
        finally
        {
            Log.CloseAndFlush();
        }
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseSerilog()
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Creates a LoggerConfiguration for configuring the Serilog logger

Serilog will write logs to the console.

This creates a Serilog logger instance on the static Log.Logger property.

Registers the SerilogLoggerFactory and connects the Log.Logger as the sole logging provider

With the Serilog logging provider configured, you can run the application and generate some logs. Every time the app generates a log, `ILogger` writes the message to the Serilog provider, which writes it to every sink. In listing 17.6 you've only configured the console sink, so the output will look something like figure 17.9. The Serilog console sink colorizes the logs more than the built-in console provider, so I find it's somewhat easier to parse the logs visually.

> **TIP** Serilog has many great features in addition to this. One of my favorites is the ability to add *enrichers*. These automatically add information to all your log messages, such as the process ID or environment variables, which can be useful when diagnosing problems. For an in-depth look at the recommended way to configure Serilog for ASP.NET Core apps, see the "Setting up Serilog in ASP.NET Core 3" post by Nicholas Blumhardt, the creator of Serilog: http://mng.bz/Yqvz.

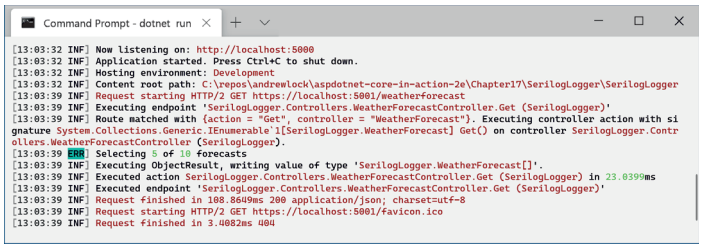


Figure 17.9 Example output using the Serilog provider and a console sink. The output has more colorization than the built-in console provider, though by default it doesn't display the log category for each log.[4]

Serilog lets you easily plug in additional sinks to your application, in much the same way as you do with the default ASP.NET Core abstractions. Whether you choose to use Serilog or stick to other providers is up to you; the feature sets are quite similar, though Serilog is more mature. Whichever you choose, once you start running your apps in production, you'll quickly discover a different issue: the sheer number of log messages your app generates!

17.4 Changing log verbosity with filtering

In this section you'll see how to reduce the number of log messages written to the logger providers. You'll learn how to apply a base level filter, filter out messages from specific namespaces, and use logging provider-specific filters.

If you've been playing around with the logging samples, you'll probably have noticed that you get a lot of log messages, even for a single request like the one in figure 17.2: messages from the Kestrel server, messages from EF Core, not to mention your own custom messages. When you're debugging locally, having access to all that detailed information is extremely useful, but in production you'll be so swamped by noise that it'll make picking out the important messages difficult.

ASP.NET Core includes the ability to filter out log messages *before* they're written, based on a combination of three things:

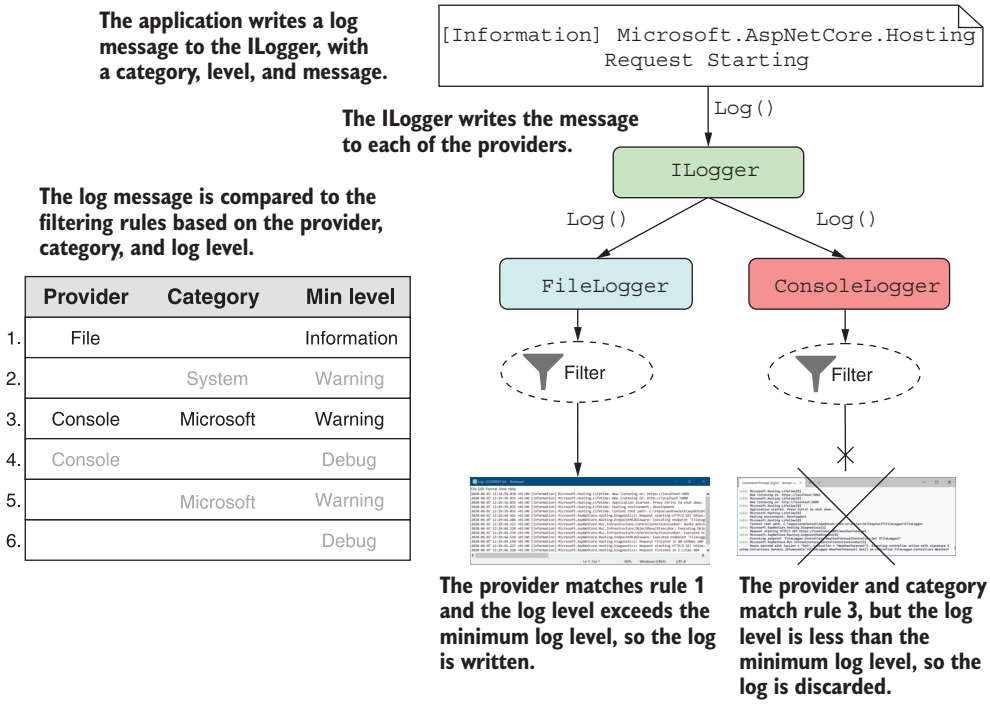
- The log level of the message
- The category of the logger (who created the log)
- The logger provider (where the log will be written)

You can create multiple rules using these properties, and, for each log that's created, the most specific rule will be applied to determine whether the log should be written to the output. You could create the following three rules:

[4] If you wish to display the log category in the console sink, you can customize the `outputTemplate` and add `{SourceContext}`. For details, see the README on GitHub: https://github.com/serilog/serilog-sinks-console#output-templates.

- *The default minimum log level is* `Information`—If no other rules apply, only logs with a log level of `Information` or above will be written to providers.
- *For categories that start with* `Microsoft`, *the minimum log level is* `Warning`—Any logger created in a namespace that starts with `Microsoft` will only write logs that have a log level of `Warning` or above. This would filter out the noisy framework messages you saw in figure 17.6.
- *For the console provider, the minimum log level is* `Error`—Logs written to the console provider must have a minimum log level of `Error`. Logs with a lower level won't be written to the console, though they might be written using other providers.

Typically, the goal with log filtering is to reduce the number of logs written to certain providers, or from certain namespaces (based on the log category). Figure 17.10 shows a possible set of filtering rules that apply to the console and file logging providers.



	Provider	Category	Min level
1.	File		Information
2.		System	Warning
3.	Console	Microsoft	Warning
4.	Console		Debug
5.		Microsoft	Warning
6.			Debug

Figure 17.10 Applying filtering rules to a log message to determine whether a log should be written. For each provider, the most specific rule is selected. If the log exceeds the rule's required minimum level, the provider writes the log; otherwise it discards it.

In this example, the console logger explicitly restricts logs written in the `Microsoft` namespace to `Warning` or above, so the console logger ignores the log message shown. Conversely, the file logger doesn't have a rule that explicitly restricts the `Microsoft`

namespace, so it uses the configured minimum level of `Information` and writes the log to the output.

> **TIP** Only a single rule is chosen when deciding whether a log message should be written; they aren't combined. In figure 17.10, rule 1 is considered more specific than rule 5, so the log is written to the file provider, even though both could technically apply.

You typically define your app's set of logging rules using the layered configuration approach discussed in chapter 11, because this lets you easily have different rules when running in development and production. You do this by calling `AddConfiguration` when configuring logging in Program.cs, but `CreateDefaultBuilder()` also does this for you automatically.

This listing shows how you'd add configuration rules to your application when configuring your own `HostBuilder`, instead of using the `CreateDefaultBuilder` helper method.

Listing 17.7 Loading logging configuration in `ConfigureLogging`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        new HostBuilder()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .ConfigureAppConfiguration(config =>
                config.AddJsonFile("appsettings.json"))
            .ConfigureLogging((ctx, builder) =>
            {
                builder.AddConfiguration(
                    ctx.Configuration.GetSection("Logging"));
                builder.AddConsole();
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Loads configuration values from appsettings.json

Loads the log filtering configuration from the Logging section and adds to ILoggingBuilder

Adds a console provider to the app

In this example, I've loaded the configuration from a single file, appsettings.json, which contains all of our app configuration. The logging configuration is specifically contained in the `"Logging"` section of the `IConfiguration` object, which is available when you call `Configurelogging()`.

> **TIP** As you saw in chapter 11, you can load configuration settings from multiple sources, like JSON files and environment variables, and can load them

conditionally based on the `IHostingEnvironment`. A common practice is to include logging settings for your production environment in appsettings.json and overrides for your local development environment in appsettings.Development.json.

The logging section of your configuration should look similar to the following listing, which shows how you could define the rules shown in figure 17.10.

Listing 17.8 The log filtering configuration section of appsettings.json

```
{
  "Logging": {
    "LogLevel": {                          Rules to apply
      "Default": "Debug",                  if there are no
      "System": "Warning",                 applicable rules
      "Microsoft": "Warning"               for a provider
    },
    "File": {                                        Rules to apply to
      "LogLevel": {                                  the File provider
        "Default": "Information"
      }
    },
    "Console": {                                     Rules to apply
      "LogLevel": {                                  to the Console
        "Default": "Debug",                          provider
        "Microsoft": "Warning"
      }
    }
  }
}
```

When creating your logging rules, the important thing to bear in mind is that if you have *any* provider-specific rules, these will take precedence over the category-based rules defined in the `"LogLevel"` section. Therefore, for the configuration defined in listing 17.8, if your app *only* uses the file or console logging providers, the rules in the `"LogLevel"` section will effectively never apply.

If you find this confusing, don't worry, so do I. Whenever I'm setting up logging, I check the algorithm used to determine which rule will apply for a given provider and category, which is as follows:

1. Select all rules for the given provider. If no rules apply, select all rules that don't define a provider (the top `"LogLevel"` section from listing 17.8).
2. From the selected rules, select rules with the longest matching category prefix. If no selected rules match the category prefix, select the `"Default"` if present.
3. If multiple rules are selected, use the last one.
4. If no rules are selected, use the global minimum level, `"LogLevel:Default"` (which is `Debug` in listing 17.8).

Each of these steps (except the last) narrows down the applicable rules for a log message, until you're left with a single rule. You saw this in effect for a `"Microsoft"` category log in figure 17.10. Figure 17.11 shows the process in more detail.

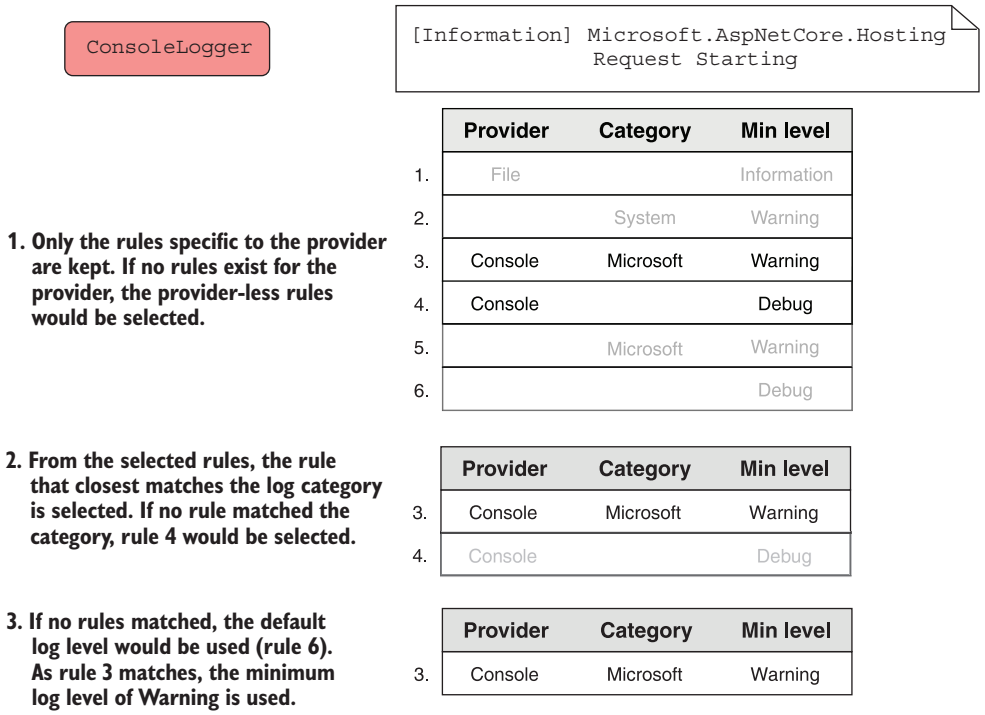


Figure 17.11 Selecting a rule to apply from the available set for the console provider and an `Information` level log. Each step reduces the number of rules that apply until you're left with only one.

> **WARNING** Log filtering rules aren't merged together; a single rule is selected. Including provider-specific rules will override global category-specific rules, so I tend to stick to category-specific rules in my apps to make the overall set of rules easier to understand.

With some effective filtering in place, your production logs should be much more manageable, as shown in figure 17.12. Generally I find it's best to limit the logs from the ASP.NET Core infrastructure and referenced libraries to `Warning` or above, while keeping logs that my app writes to `Debug` in development and `Information` in production.

This is close to the default configuration used in the ASP.NET Core templates. You may find you need to add additional category-specific filters, depending on which

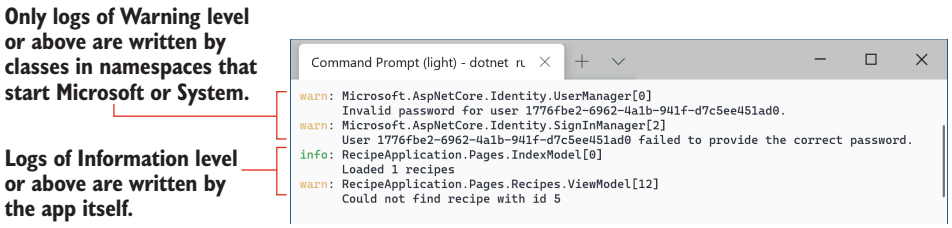


Figure 17.12 Using filtering to reduce the number of logs written. In this example, category filters have been added to the Microsoft and System namespaces, so only logs of Warning and above are recorded. That increases the proportion of logs that are directly relevant to your application.

NuGet libraries you use and the categories they write to. The best way to find out is generally to run your app and see if you get flooded with uninteresting log messages.

Even with your log verbosity under control, if you stick to the default logging providers like the file or console loggers, you'll probably regret it in the long run. These log providers work perfectly well, but when it comes to finding specific error messages, or analyzing your logs, you'll have your work cut out for you. In the next section you'll see how structured logging can help tackle this problem.

17.5 Structured logging: Creating searchable, useful logs

In this section you'll learn how structured logging makes working with log messages easier. You'll learn to attach key-value pairs to log messages and how to store and query for key values using the structured logging provider, Seq. Finally, you'll learn how to use scopes to attach key-value pairs to all log messages within a block.

Let's imagine you've rolled out the recipe application we've been working on to production. You've added logging to the app so that you can keep track of any errors in your application, and you're storing the logs in a file.

One day, a customer calls and says they can't view their recipe. Sure enough, when you look through the log messages, you a see a warning:

```
warn: RecipeApplication.Controllers.RecipeController[12]
      Could not find recipe with id 3245
```

This piques your interest—why did this happen? Has it happened before for this *customer*? Has it happened before for this *recipe*? Has it happened for *other* recipes? Does it happen regularly?

How would you go about answering these questions? Given that the logs are stored in a text file, you might start doing basic text searches in your editor of choice, looking for the phrase `"Could not find recipe with id"`. Depending on your notepad-fu skills, you could probably get a fair way in answering your questions, but it would likely be a laborious, error-prone, and painful process.

The limiting factor is that the logs are stored as *unstructured* text, so text processing is the only option available to you. A better approach is to store the logs in a *structured* format, so that you can easily query the logs, filter them, and create analytics. Structured logs could be stored in any format, but these days they're typically represented as JSON. For example, a structured version of the same recipe warning log might look something like this:

```
{
  "eventLevel": "Warning",
  "category": "RecipeApplication.Controllers.RecipeController",
  "eventId": "12",
  "messageTemplate": "Could not find recipe with {recipeId}",
  "message": "Could not find recipe with id 3245",
  "recipeId": "3245"
}
```

This structured log message contains all the same details as the unstructured version, but in a format that would easily let you search for specific log entries. It makes it simple to filter logs by their `EventLevel`, or to only show those logs relating to a specific recipe ID.

> **NOTE** This is only an example of what a structured log could look like. The format used for the logs will vary depending on the logging provider used and could be anything. The key point is that properties of the log are available as key-value pairs.

Adding structured logging to your app requires a logging provider that can create and store structured logs. Elasticsearch is a popular general search and analytics engine that can be used to store and query your logs. Serilog includes a sink for writing logs to Elasticsearch that you can add to your app in the same way as you added the console sink in section 17.3.2.

> **TIP** If you want to learn more about Elasticsearch, *Relevant Search* by Doug Turnbull and John Berryman (Manning, 2016) is a great choice. It will show how you can make the most of your structured logs.

Elasticsearch is a powerful production-scale engine for storing your logs, but setting it up isn't for the faint of heart. Even after you've got it up and running, there's a somewhat steep learning curve associated with the query syntax. If you're interested in something more user-friendly for your structured logging needs, Seq (https://getseq.net) is a great option. In the next section I'll show you how adding Seq as a structured logging provider makes analyzing your logs that much easier.

17.5.1 Adding a structured logging provider to your app

To demonstrate the advantages of structured logging, in this section you'll configure an app to write logs to Seq. You'll see that the configuration is essentially identical to unstructured providers, but that the possibilities afforded by structured logging make considering it a no-brainer.

Seq is installed on a server or your local machine and collects structured log messages over HTTP, providing a web interface for you to view and analyze your logs. It is currently available as a Windows app or a Linux Docker container. You can install a free version for development, which will allow you to experiment with structured logging in general.[5]

From the point of view of your app, the process for adding the Seq provider should be familiar:

1. Install the Seq logging provider using Visual Studio or the .NET CLI with

```
dotnet add package Seq.Extensions.Logging
```

2. Add the Seq logging provider in Program.cs inside the `ConfigureLogging` method. To add the Seq provider in addition to the console and debug providers included as part of `CreateDefaultBuilder`, use

```
Host.CreateDefaultBuilder(args)
    .ConfigureLogging(builder => builder.AddSeq())
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
```

That's all you need to add Seq to your app. This will send logs to the default local URL when you have Seq installed in your local environment. The `AddSeq()` extension method includes additional overloads to customize Seq when you move to production, but this is all you need to start experimenting locally.

If you haven't already, install Seq on your development machine and navigate to the Seq app at http://localhost:5341. In a different tab, open up your app and start browsing around and generating logs. Back in Seq, if you refresh the page, you'll see a list of logs, something like in figure 17.13. Clicking on a log expands it and shows you the structured data recorded for the log.

ASP.NET Core supports structured logging by treating each captured parameter from your message format string as a key-value pair. If you create a log message using the following format string,

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

the logging provider will create a `RecipeCount` parameter with a value of `Recipes.Count`. These parameters are added as properties to each structured log, as you can see in figure 17.13.

[5] You can download the Windows installer for Seq from https://getseq.net/Download.

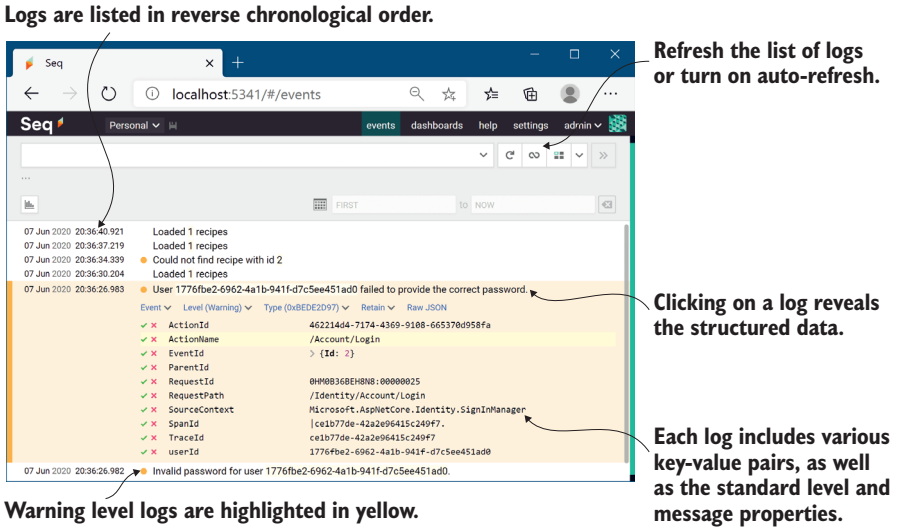


Figure 17.13 The Seq UI. Logs are presented as a list. You can view the structured logging details of individual logs, view analytics for logs in aggregate, and search by log properties.

Structured logs are generally easier to read than your standard-issue console output, but their real power comes when you need to answer a specific question. Consider the problem from before, where you see this error:

```
Could not find recipe with id 3245
```

You want to get a feel for how widespread the problem is. The first step would be to identify how many times this error has occurred and to see whether it's happened to any other recipes. Seq lets you filter your logs, but it also lets you craft SQL queries to analyze your data, so finding the answer to the question takes a matter of seconds, as shown in figure 17.14.

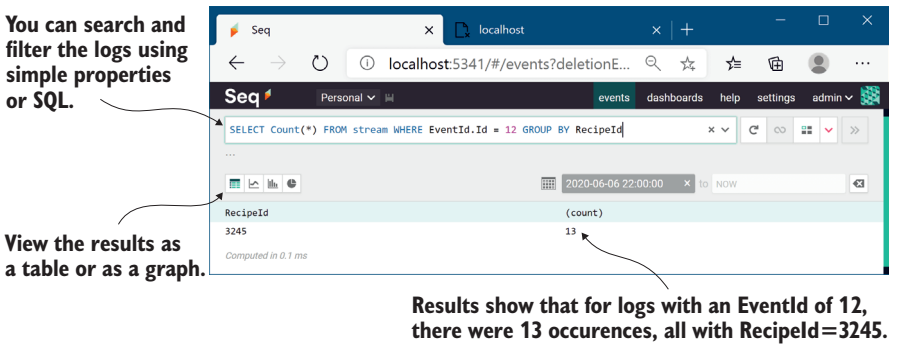


Figure 17.14 Querying logs in Seq. Structured logging makes log analysis like this example easy.

NOTE You don't need query languages like SQL for simple queries, but it makes digging into the data easier. Other structured logging providers may provide query languages other than SQL, but the principle is the same as in this Seq example.

A quick search shows that you've recorded the log message with `EventId.Id=12` (the `EventId` of the warning we're interested in) 13 times, and every time the offending `RecipeId` was 3245. This suggests that there may be something wrong with that recipe in particular, which points you in the right direction to find the problem.

More often than not, figuring out errors in production involves logging detective work like this to isolate where the problem occurred. Structured logging makes this process significantly easier, so it's well worth considering, whether you choose Seq, Elasticsearch, or a different provider.

I've already described how you can add structured properties to your log messages using variables and parameters from the message, but as you can see in figure 17.13, there are far more properties visible than exist in the message alone.

Scopes provide a way to add arbitrary data to your log messages. They're available in some unstructured logging providers, but they shine when used with structured logging providers. In the final section of this chapter, I'll demonstrate how you can use them to add additional data to your log messages.

17.5.2 Using scopes to add additional properties to your logs

You'll often find in your apps that you have a group of operations that all use the same data, which would be useful to attach to logs. For example, you might have a series of database operations that all use the same transaction ID, or you might be performing multiple operations with the same user ID or recipe ID. *Logging scopes* provide a way of associating the same data to every log message in such a group.

DEFINITION *Logging scopes* are used to group multiple operations by adding the same data to each log message.

Logging scopes in ASP.NET Core are created by calling `ILogger.BeginScope<T>(T state)` and providing the `state` data to be logged. You create scopes inside a `using` block; any log messages written inside the scope block will have the associated data, whereas those outside won't.

Listing 17.9 Adding scope properties to log messages with `BeginScope`

```
_logger.LogInformation("No, I don't have scope");

using(_logger.BeginScope("Scope value"))
using(_logger.BeginScope(new Dictionary<string, object>
    {{ "CustomValue1", 12345 } }))
{
    _logger.LogInformation("Yes, I have the scope!");
}

_logger.LogInformation("No, I lost it again");
```

Log messages written outside the scope block don't include the scope state.

Calling BeginScope starts a scope block, with a scope state of "Scope value".

You can pass anything as the state for a scope.

Log messages written inside the scope block include the scope state.

The scope state can be any object at all: an `int`, a `string`, or a `Dictionary`, for example. It's up to each logging provider implementation to decide how to handle the state you provide in the `BeginScope` call, but typically it will be serialized using `ToString()`.

> **TIP** The most common use for scopes I've found is to attach additional key-value pairs to logs. To achieve this behavior in Seq and Serilog, you need to pass `Dictionary<string, object>` as the state object.[6]

When the log messages inside the scope block are written, the scope state is captured and written as part of the log, as shown in figure 17.15. The `Dictionary<>` of key-value pairs is added directly to the log message (`CustomValue1`), and the remaining state values are added to the `Scope` property. You will likely find the dictionary approach the more useful of the two, as the added properties are more easily filtered on, as you saw in figure 17.14.

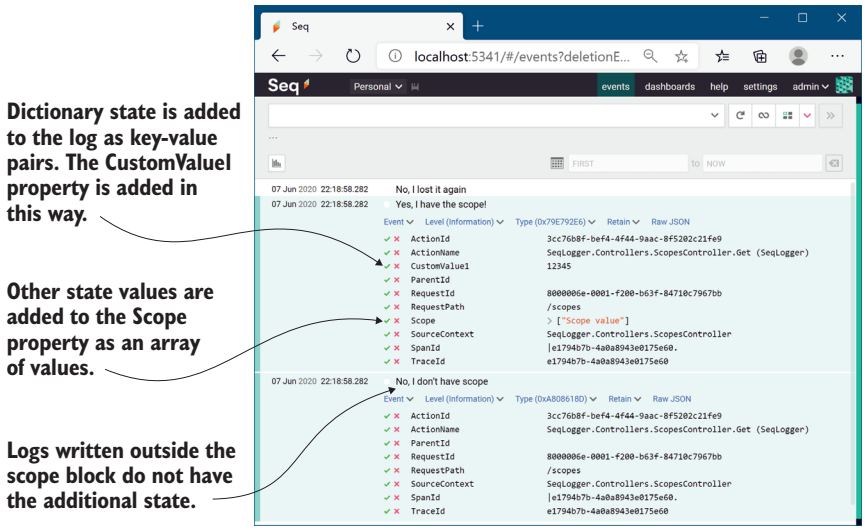


Figure 17.15 Adding properties to logs using scopes. Scope state added using the dictionary approach is added as structured logging properties, but other state is added to the `Scope` property. Adding properties makes it easier to associate related logs with one another.

That brings us to the end of this chapter on logging. Whether you use the built-in logging providers or opt to use a third-party provider like Serilog or NLog, ASP.NET Core makes it easy to get detailed logs not only for your app code, but for the libraries

[6] Nicholas Blumhardt, the creator of Serilog and Seq, has examples and the reasoning for this on his blog in the "The semantics of ILogger.BeginScope()" article: http://mng.bz/GxDD.

that make up your app's infrastructure, like Kestrel and EF Core. Whichever you choose, I encourage you to add more logs than you *think* you'll need—future-you will thank me when it comes to tracking down a problem.

In the next chapter, we'll look in detail at a variety of web security problems that you should consider when building your apps. ASP.NET Core takes care of some of these issues for you automatically, but it's important to understand where your app's vulnerabilities lie, so you can mitigate them as best you can.

Summary

- Logging is critical to quickly diagnosing errors in production apps. You should always configure logging for your application so that logs are written to a durable location.
- You can add logging to your own services by injecting `ILogger<T>`, where `T` is the name of the service. Alternatively, inject `ILoggerFactory` and call `CreateLogger()`.
- The log level of a message indicates how important it is and ranges from `Trace` to `Critical`. Typically you'll create many low-importance log messages and a few high-importance log messages.
- You specify the log level of a log by using the appropriate extension method of `ILogger` to create your log. To write an `Information` level log, use `ILogger.LogInformation(message)`.
- The log category indicates which component created the log. It is typically set to the fully qualified name of the class creating the log, but you can set it to any string if you wish. `ILogger<T>` will have a log category of `T`.
- You can format messages with placeholder values, similar to the `string.Format` method, but with meaningful names for the parameters. Calling `logger.LogInfo("Loading Recipe with id {RecipeId}", 1234)` would create a log reading `"Loading Recipe with id 1234"`, but it would also capture the value `RecipeId=1234`. This *structured logging* makes analyzing log messages much easier.
- ASP.NET Core includes many logging providers out of the box. These include the console, debug, EventLog, and EventSource providers. Alternatively, you can add third-party logging providers.
- You can configure multiple `ILoggerProvider` instances in ASP.NET Core, which define where logs are output. The `CreateDefaultBuilder` method adds the console and debug providers, and you can add additional providers by calling `ConfigureLogging()`.
- Serilog is a mature logging framework that includes support for a large number of output locations. You can add Serilog to your app with the Serilog.AspNetCore package. This replaces the default `ILoggerFactory` with a Serilog-specific version.
- You can control logging output verbosity using configuration. The `CreateDefaultBuilder` helper uses the `"Logging"` configuration section to control

output verbosity. You typically will filter out more logs in production compared to when developing your application.

- Only a single log filtering rule is selected for each logging provider when determining whether to output a log message. The most specific rule is selected based on the logging provider and the category of the log message.
- Structured logging involves recording logs so that they can be easily queried and filtered, instead of the default unstructured format that's output to the console. This makes analyzing logs, searching for issues, and identifying patterns easier.
- You can add additional properties to a structured log by using scope blocks. A scope block is created by calling `ILogger.BeginScope<T>(state)` in a `using` block. The state can be any object and is added to all log messages inside the scope block.